Iteradores, *Templates* e Consultas na Análise e Manipulação de Programas

Ademir A. Oliveira[1] **, Rodrigo G. Ribeiro**[2] **, Denis P. Pinheiro**[2] **,**
Thiago H. Braga[1] **, Marcelo A. Maia**[3] **, Roberto S. Bigonha**[2]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Campus da Pampulha – 31270-010 Belo Horizonte, MG

[2]Departamento de Computação – Universidade Federal de Ouro Preto
Campus Morro do Cruzeiro – 35400-000 Ouro Preto, MG

[3]Faculdade de Computação – Universidade Federal de Uberlândia
Campus Santa Mônica – 38400-902 Uberlândia, MG

{ademirao,thiagohb,bigonha}@dcc.ufmg.br
{rodrigogribeiro,denisppinheiro}@iceb.ufop.br
marcmaia@facom.ufu.br

Abstract. *A engenharia reversa e a reestruturação de programas são utéis em tarefas de manutenção e evolução de software. As ferramentas e meta-ferramentas para análise e manipulação de código-fonte têm um papel importante nestas atividades, contudo seu desenvolvimento é difícil dada à complexidade intrínsica desta classe de ferramentas. A existência de meta-ferramentas melhores poderia facilitar a construção das ferramentas para análise e manipulação de código. Apesar de diversas meta-ferramentas já terem sido propostas, ainda não existe uma solução definitiva. Este trabalho propõe o uso integrado de iteradores, templates, e consultas relacionais para facilitar a construção de módulos para análise e manipulação de programas. Estes mecanismos são baseados nas noções de referências tipadas para código-fonte e de casamento de padrões em árvores sintáticas. Para demonstrar a efetividade desta abordagem, apresenta-se um construtor de grafos de semântica estática baseada em declarações, escopo e uso de símbolos. Para avaliar a performance das abordagens iterativas e declarativas, foram implementadas e analisadas duas refatorações. As meta-ferramentas apresentadas neste artigo apresentaram vantagens sobre outras meta-ferramentas, portanto podendo ser consideradas viáveis de serem introduzidas em processos de manutenção de software.*

1. Introdução

A manutenção e evolução de software têm sistematicamente trazido novos desafios para os desenvolvedores de software. Sistemas de software estão cada vez mais complexos, e durante o processo de manutenção, a reestruturação sistemática de código tem sido amplamente adotada, favorecida pelo desenvolvimento de ambientes integrados de desenvolvimento com suporte a refatorações[Mens and Tourwé, 2004]. Este contexto pode ainda ser ampliado considerando-se o desafio de se extrair aspectos a partir de sistemas OO, para favorecer a modularização. Neste cenário, são bem-vindas ferramentas para auxiliar desenvolvedores manipular código-fonte sistematicamente. Infelizmente, estas ferramentas muitas vezes não se adequam a perfis específicos de

diferentes projetos de software em diferentes empresas. Além disso, a construção de ferramentas de análise e manipulação de código adaptadas à uma equipe ou projeto específicos é extremante difícil e cara. O estudo de meta-ferramentas para auxiliar na construção de ferramentas de análise e manipulação já foi explorado na literatura [Johnson, 1975, Cordy et al., 2002, van den Brand and et.al., 2001]. Contudo, a adoção destas ferramentas por equipes de desenvolvimento ainda é baixa, talvez porque algumas delas sejam de "baixo nível", como o Yacc, ou porque requeiram o aprendizado de um paradigma baseado em sistemas de reescrita, como TXL e ASF+SDF, incomum a grande parte dos desenvolvedores.

O objetivo deste trabalho é mostrar a efetiva integração de alguns conceitos de meta-ferramentas bem conhecidos em uma linguagem orientada a objetos. Estes conceitos são iteradores e consultas relacionais aplicadas em referências de código-fonte tipado, e *templates* para casamento de padrões em árvores de sintaxe.

Este artigo usa o ambiente MetaJ para prover iteradores e *templates*, e usa a linguagem SCQL para prover consultas do tipo SQL[Oliveira, 2004, Oliveira et al., 2004]. Os mecanismos propostos podem ser divididos em duas classes: (i) declarativos, tais como, *templates* e consultas, e (ii) imperativos, tais como, iteradores.

Este artigo apresenta questões relativas ao uso conjunto de mecanismos declarativos e imperativos de meta-programação, e compara estes respectivos mecanismos entre si. São apresentados meta-programas desenvolvidos com *templates* e iteradores de MetaJ e com consultas SCQL. MetaJ e SCQL foram escolhidos porque são compatíveis entre si e podem ser usados em um mesmo programa. Caberá ao programador decidir qual dos dois é mais apropriado em determinada situação.

A próxima seção apresenta o ambiente MetaJ. A Seção 3 apresenta a linguagem SCQL. Na Seção 4 são apresentados os trabalhos relacionados. Na Seção 5, com o objetivo de demonstrar a funcionalidade de MetaJ e SCQL, é apresentado um estudo de caso que mostra o uso conjunto de ambas as ferramentas para construir um modelo de programas Java apropriado à análise baseada em semântica estática. Na Seção 6, com o objetivo de comparar as abordagens declarativa (SCQL) e imperativa (MetaJ), são apresentados resultados comparativos sobre a implementação de refatorações. Na Seção 7, discute-se os resultados obtidos e finalmente apresenta-se considerações finais.

2. O Ambiente MetaJ

MetaJ é um ambiente de meta-programação orientado a objetos que é implementado como uma extensão da linguagem Java. Tem quatro conceitos básicos para meta-programação: referências para programas (referências-p), *templates* de programas, iteradores em programas e plug-ins específicos a uma linguagem. Os meta-programas MetaJ são programas Java que usam implementações destes conceitos, o que permite ao programador usar todo conhecimento sobre desenvolvimento de programas orientados a objetos. Veja abaixo um exemplo de programa MetaJ.

```
// importing p-reference API and a user defined template
import metaj.framework.PReference;
import myTemplates.SelectPackageName;
public class Main {
   public static void main (...) throws ...{
      // Criando uma referencia-p tipada
      PReference r = MetaJSystem.createPReference("java","#compilation_unit");
      // Atualizando o valor de uma referencia-p
      r.setFile("/samples/HelloWorld.java");
      // Usando um template definido pelo usuario
      SelectPackageName spn = new SelectPackageName ();
      // Verificando se o codigo casa com o padrao definido no template spn
```

```
      if(spn.match(r)){
         // Obtem um iterador para explorar o nome do pacote
         Iterator it = spn.getPackageName().getIterator();
         // Iteracao para acessar cada identificador do nome do pacote
         while(it.hasNext()){
            it.next();
            ...
         }
      }else{ ... // error }
  }
}
```

O código acima mostra um programa MetaJ simples que manipula um programa Java que está em um arquivo /samples/HelloWorld.java. Inicialmente, é criada uma referência-p cujo valor é todo o conteúdo do arquivo. Depois, uma instância de um *template* definido pelo usuário (veja na Seção 2.3) é usado para selecionar o nome do pacote declarado no programa. Finalmente, é criado um iterador a partir do trecho referente ao nome do pacote e o mesmo utilizado para explorar cada identificador que ocorre neste nome. É importante salientar o uso implícito de plug-ins dependentes de Java. Neste caso o meta-programa manipula programas Java. Quando uma referência-p ou *template* é criado, o nome do plug-in deve ser fornecido para ativar o respectivo *parser*. No exemplo acima, quando a referência-p `r` é criada, o nome do plug-in (`java`) é passado como o primeiro parâmetro do método `MetaJSystem.createPReference`. As principais características de MetaJ são apresentadas nas próximas seções. Maiores informações sobre MetaJ podem ser encontradas em [Oliveira, 2004, Oliveira et al., 2004].

2.1. Referências a Programas

Referências a programas (referências-p) são abstrações que armazenam trechos de código-fonte. Elas escondem a representação interna do código, e permitem apenas operações que garantam a preservação da consistência sintática do código manipulado. Elas são tipadas com o tipo do nó raiz de sua árvore sintática correspondente. Os tipos dos nós das árvores sintáticas são extraídos dos não-terminais da gramática da linguagem. Os plug-ins, que são responsáveis por esta funcionalidade para uma gramática específicas, são gerados automaticamente a partir de uma especificação de gramática.

O resumo da API para referências-p é mostrada na Tabela 1.

Método	Funcionalidade
`set, setFile, setDeref,` `add, remove, replace`	modifica o valor de uma referência-p
`match, equals,contains,` `hasType, isComposedBy,`	compara e verifica o valor de uma referência-p
`duplicate, toString,` `get, getSize`	duplica, converte para string ou recupera valores
`getIterator`	retorna um iterador para o respectivo código

Tabela 1: Métodos para a abstração *referências-p*

O último método na Tabela 1 retorna um iterador, descrito na próxima seção.

2.2. Iteradores

Iteradores são objetos usados para percorrer o código-fonte armazenado em uma referência-p. Tais objetos encapsulam um caminhamento *top-down* iterativo, iniciando na raiz da árvore. A interface *Iterator* provê métodos que permitem o controle do caminhamento. Os métodos principais da interface são mostrados na Tabela 2.

Método	Funcionalidade
`boolean hasNext()`	verifica se existe algum trecho de código a ser alcançado no próximo passo
`void nextIn()`	alcança o próximo trecho de código, em pré-ordem .
`PReference getPReference`	retorna uma nova referência-p com o trecho de código alcançado na última chamada do método `nextIn`.

Tabela 2: Principais métodos da interface *Iterator*

2.3. *Templates*

Um *template* é uma abstração que encapsula um padrão de programa usado para compor e decompor trechos de programas. Um padrão é uma sentença da linguagem-fonte (linguagem do programa sendo manipulado) onde uma meta-anotação pode aparecer no lugar de uma construção sintática. O padrão é composto por dois tipos de meta-anotações: meta-variáveis e marcadores de trechos opcionais. Toda meta-anotação tem um tipo associado, correspondendo à estrutura sintática da linguagem-fonte. Os padrões provêem duas operações básicas: *match*, a qual verifica se um trecho de código casa com o padrão, e *print*, a qual constrói um trecho de programa baseado na estrutura do padrão. As estruturas sintáticas disponíveis para o meta-programador são selecionadas a partir dos não-terminais da gramática, e coincidem com os tipos das referências-p.

Todo símbolo de tipo deve ser prefixado com o símbolo de *anti-quotation* (#). A seguir, apresenta-se uma declaração de *template* para a linguagem Java.

```
// package declaration
package  metaj.examples.basicTemplates;
language java;  //  plug-in name (language for template)
template #compilation_unit  SelectPackageName #{
    package #name pack;
    #import_declaration_opt[ #import_declarations imps ]#
    #type_declarations tds
}#
```

Este *template* casa com qualquer programa Java (*compilation_unit*) que tenha necessariamente uma declaração de pacote e uma declaração de tipo.

Repare as declarações de meta-variáveis `#name pack`, `#type_declarations tds`, e na sentença opcional `#import_declaration_opt[...]#`. Esta última permite casamentos onde declarações de importação não estejam presentes.

Para permitir que todas características de um *template* (meta-variáveis, operações de casamento e impressão) estejam disponíveis para um meta-programa, uma declaração de *template* é compilada em uma classe Java usando o compilador de templates de MetaJ. Após a compilação é gerada uma classe com mesmo nome e pacote do *template*, a qual provê os métodos mostrados na Tabela 3.

3. A linguagem SCQL

SCQL é uma linguagem declarativa de consulta a código-fonte definida a partir de conceitos MetaJ, tais como, referências-p e estruturas sintáticas tipadas. A linguagem permite escrever consultas tipo-SQL para selecionar ocorrências de estruturas sintáticas no código-fonte, p.ex., identificadores, declarações de variáveis, declarações de classes, etc.

O projeto de SCQL foi inspirado nos conceitos de SQL. A árvore sintática do programa-fonte é visualizada como uma tabela, permitindo assim consultas relacionais. O operador VIEW TREE é responsável por esta visualização e pode ser comparado à

Método	Funcionalidade

`boolean match(PReference),`
`boolean match(String),`
`boolean matchFile(String)`
	verifica se o código passado como parâmetro casa o *template*. Esta operação atribui valores às meta-variáveis do *template*.

`setXXX(String),` `setXXX(PReference)`	define o valor da meta-variável XXX.
`PReference getXXX()`	retorna uma referência-p para o código atribuído à meta-variável XXX.
`String toString(),` `toFile(String),`	constrói um programa e o retorna como um arquivo, uma String, ou uma referência-p

Tabela 3: A API para templates

operação CREATE TABLE do SQL. O operador SELECT permite projeções, seleções e junções nas tabelas. Os operadores INSERT, UPDATE e DELETE modificam o conteúdo do código-fonte referenciado em um conjunto-resultado.

Foi construído um interpretador interativo para consultas e também uma API que permite programas MetaJ executar consultas de maneira similar a uma programa Java executando consultas SQL com JDBC.

Apresentamos a seguir os principais elementos de SCQL. Uma consulta VIEW TREE retorna uma visualização em tabela de um trecho de programa. A consulta abaixo visualiza a referência-p $prog$ como uma tabela com uma coluna lvd.

```
VIEW TREE prog
    AS TABLE #local_variable_declaration lvd
```

A coluna lvd é preenchida com todas as construções sintáticas do tipo `#local_variable_declaration`, i.e., com todas as declarações de variáveis locais.

A consulta abaixo visualiza o programa $prog$ como uma tabela com duas colunas, lvd and t.

```
VIEW TREE prog
    AS TABLE #local_variable_declaration lvd, #type t
    WHERE lvd.isComposedBy(t)
```

Cada linha desta tabela é preenchida com um par (lvd, t), onde lvd é uma declaração de variável local, t é o uso de um tipo e t compõe lvd sintaticamente. A condição "t compõe lvd" é especificada por um predicado $lvd.isComposedBy(t)$. SCQL calcula os valores a serem preenchidos na tabela combinando, através de produto cartesiano, todas as construções do tipo `#type` com construções do tipo `#local_variable_declaration`. Esta combinação gera vários pares (lvd, t), mas somente aqueles que satisfazem a condição da cláusula WHERE são inseridos na tabela.

O comando VIEW TREE provê operadores FILTERED BY, FOCUSED ON e EXCLUDING que permitem a otimização de uma consulta filtrando valores a serem inseridos em uma coluna, focalizando e evitando trechos de tipos específicos, respectivamente.

A operação SELECT permite a combinação de consultas para construir uma nova tabela. Esta tabela é preenchida com tuplas geradas pela combinação de resultados de sub-consultas. Pode ser especificada uma cláusula WHERE para filtrar somente tuplas desejadas. O exemplo abaixo junta declarações de variáveis locais que ocorrem em `prog1` com declarações de tipo `td` que ocorrem em `prog2`.

```
SELECT lvd, t, td, id FROM
    VIEW TREE prog1
    AS TABLE #local_variable_declaration lvd, #type t
```

```
    WHERE lvd.isComposedBy(t),

    VIEW TREE prog2
    AS TABLE #type_declaration td, #identifier id
    WHERE outer.verifier.isTypeName(td,id)
WHERE t.match(id)
```

O método `isTypeName(PReference, PReference)` do objeto `outer.verifier` verifica se o identificador `id` é igual ao identificador de tipo em `td`. Este método pode ser facilmente implementado com *templates* e referências-p.

Como exemplo, considere `prog1` e `prog2`, mostrados abaixo.

prog1:
```
public class Test {
  int x;
  Test (int x) { this.x = x;}
  void calc() {
    Util u = new Util();
    Other z1;
    System.out.println(u.fat(x));
  }
  public static void main(...) {
        Other z2;
        new Test(10).calc();
  }
}
```

prog2:
```
class Util {
  int fat (int x){
    if(x == 0) return 1;
    else return x*fat(x-1);
}

interface Other{
  int test();
}
```

Primeiro, calcula-se os resultados de cada VIEW TREE.

lvd:#local_variable_declaration	t:#type
Util u = new Util()	Util
Other z1	Other
Other z2	Other

td:#type_declaration	id:#identifier
class Util ...	Util
interface Other ...	Other

As entradas destas tabelas são combinadas para produzir o conjunto-resultado final: uma tabela com quatro colunas e três linhas. Cada coluna tem um nome e um tipo que corresponde à referência-p para o respectivo trecho de programa.

lvd:#loc...	t:#type	td:#type_decl...	id:#i...
Util u = new Util()	Util	class Util{...}	Util
Other z1	Other	interface Other{...}	Other
Other z2	Other	interface Other{...}	Other

Os conjuntos-resultados de um SELECT podem ser combinados em outras consultas, permitindo o aninhamento de consultas. Contudo, os aninhamentos devem ser cuidadosos dado o custo computacional de um produto cartesiano.

Maiores detalhes sobre as operações UPDATE, INSERT e DELETE podem ser encontradas em [Oliveira, 2004].

4. Trabalhos Relacionados

O casamento de padrões já foi amplamente adotado em linguagens de programação funcionais. Em [Sellink and Verhoef, 1998], é usado casamento de padrões baseados na gramática da linguagem para análise de código-fonte, aplicado a ASF+SDF. Os templates de MetaJ aplicam-se a uma linguagem OO e as meta-variáveis são tipadas.

Também já foram propostas ferramentas de análise de código-fonte baseadas em iteradores e consultas. Um critério de comparação das ferramentas com a nossa é em relação ao modelo de código-fonte subjacente. Nosso modelo é baseado somente na sintaxe da linguagem, o que nos permitiu a automatização da construção de plugins a partir da gramática da linguagem. Modelos baseados na semântica, apesar de serem imediamente mais expressivos, tornam-se mais dependentes da linguagem a ser manipulada. JJTraveler é um framework Java usado para combinar *visitors* para analisar código-fonte [van Deursen and Visser, 2004]. O framework é gerado com JJForester a partir de especificações SDF da gramática da linguagem. Esta abordagem é baseada na visita iterativa aos nós da árvore sintática para efetuar a análise do código. Astlog é uma variante do Prolog para localizar e analisar artefatos sintáticos em árvores de sintaxe abstrata C/C++[Crew, 1997]. Astlog introduz várias características ad-hoc adaptadas a C/C++. Nossa abordagem, mesmo manipulando somente arquivos Java neste artigo, não está atrelada a aspectos semânticos de Java e portanto pode ser aplicada muito mais facilmente a qualquer outra linguagem. Genoa é uma ferramenta de análise de código baseada na noção de caminhamento em grafos de semântica abstrata[Devanbu, 1999], o que difere da nossa proposta, tanto na noção de somente caminhar iterativamente em uma estrutura, como também, por considerar informações semânticas da linguagem subjacente. Genoa se aplica para análise, enquanto nossa abordagem também se aplica na manipulação de programas. PQL é uma linguagem de consulta para programas baseada na idéia de modelar várias informações sobre os mesmos para responder às consultas[Jarzabek, 1998]. O modelo do programa é a parte mais difícil de ser construída pois requer várias análises semânticas. Os conjuntos-resultado não são baseados em tabelas, tal como SCQL, mas sim em tuplas de listas. SCA é uma álgebra de código-fonte que permite aos usuários expressarem consultas e visões complexas do código-fonte com expressões algébricas[Paul and Prakash, 1996]. Consultas em SCA são calculadas sobre um modelo de dados baseado em objetos e dependente da semântica da linguagem fonte, o que também difere de nossa proposta. JQuery é uma linguagem que incrementa Tyruba, a qual é uma linguagem de programação lógica, com uma biblioteca de predicados pré-definidos para consultas em unidade de código-fonte e em relações entre as unidades[Janzen and de Volder, 2003]. Os predicados implementam relacionamentos semânticos entre as unidades, por exemplo, quais métodos chamam outros. Graphlog é uma linguagem lógica de consulta para visualizar e pesquisar sistemas de software modelados como grafos dirigidos[Consens et al., 1992]. A construção do grafo involve informação semântica da linguagem fonte. As consultas são construídas desenhando padrões de grafos com um editor gráficos. Maior parte da informação a ser pesquisada é baseada em relacionamentos de dependência. Omega é um ambiente onde todas as informações calculadas a partir de programas são armazenadas em tabelas usando um banco de dados relacional[Linton, 1984]. O modelo relacional subjacente considera apenas linguagens procedimentais e usa tanto informações sintáticas quanto semânticas. Horwitz define um modelo para adicionar facilidade de consulta relacional aos ambientes de desenvolvimento de software[Horwitz, 1990]. O modelo se baseia no uso de relações implícitas, as quais não são armazenadas como um conjunto de tuplas, mas computadas sob necessidade during o cálculo de uma consulta. Esta abordagem é similar à nossa para cálculo de consultas. O modelo também se baseia em várias funções ad-hoc para extrair

a informação desejada de sistemas de software. Estas funções variam de fecho transitivo de chamadas de funções até informações dinâmicas de variáveis durante a execução.

5. Estudo de caso: grafo de semântica estática

Esta seção apresenta um estudo de caso para motivar a utilidade de MetaJ e SCQL. Transformações complexas em programas, tais como refatorações, podem ser muito mais facilmente escritas se houverem abstrações mais expressivas que uma árvore sintática. Por exemplo, a refatoração *Rename Class* de programas Java requer a atualização, por todo sistema, de todas ocorrências do nome alterado, seja em uma cláusula *extends*, ou em uma criação de objeto, ou em um declarador de variável, ou mesmo em uma definição de classe anônima. Para encontrar todas estas ocorrências em uma árvore de sintaxe é necessário mais que simplesmente percorrer nós de árvores e aplicar alterações de nome. Para o caso geral, é também necessário considerar questões como escopo e visibilidade.

A seguir é apresentado um meta-programa que constrói um modelo que captura semântica estática de código-fonte Java. O projeto deste modelo foi influenciado pela sua utilidade na especificação de procedimentos de refatorações. O modelo é um grafo onde seus nós são declarações, escopos e uso de símbolos. Este grafo também está ligado à respectiva árvore sintática do código. Um nó de escopo define uma área onde declarações e usos estão localizados. Cada escopo tem três conjuntos: um conjunto de todas as declarações dentro do escopo, um conjunto com todas ocorrências de símbolos dentro do escopo e um conjunto de todos escopos diretamente aninhados. Os nós de declaração são definidos por um identificador e um tipo (seja o tipo de uma variável, uma assinatura de método, ou um tipo propriamente declarado). A partir de um nó de declaração é possível navegar para todos seus *nós de uso* correspondentes. Esta características diferencia este modelo de tabelas de símbolos convencionais. Algumas declarações podem abrir um novo escopo, sejam, um corpo de classe, um corpo de método ou uma declaração de variável local. Os nós de uso representam as ocorrências de um símbolo previamente declarado. A partir de um nó de uso é possível navegar para seu nó de declaração correspondente.

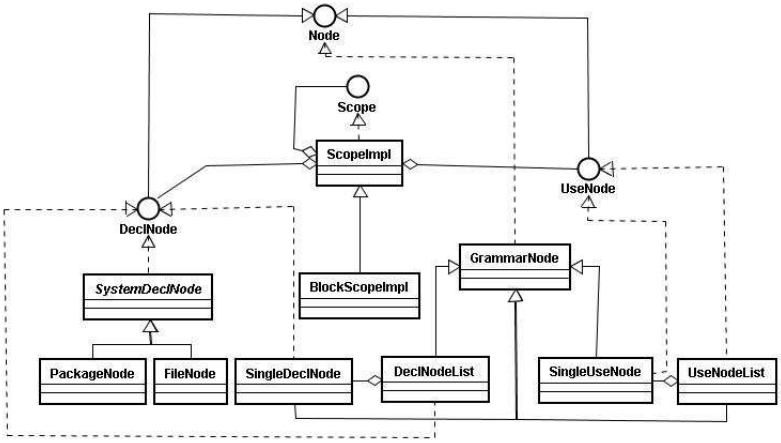


Figura 1: Modelo de semântica estática

A Figura 1 mostra as classes e interfaces usadas para representar o modelo de semântica estática. A interface `Node` define operações comuns a todos nós de declaração e uso. A classe `GrammarNode` representa nós que têm referências para código-fonte, i.e., referências-p de MetaJ.

A interface `DeclNode` define operações comuns a todos os tipos de nós de declaração: o `SystemDeclNode` que modela elementos dos sistema de ar-

quivos (`PackageNode` e `FileNode`), os quais não estão definidos na gramática; o `SingleDeclNode` que modela uma declaração única; e o `DeclNodeList` que modela uma lista de declarações que têm o mesmo tipo, que são declaração de variáveis de instância e variáveis locais.

A interface `UseNode` define operações comuns a dois tipos de nós de uso: o `SingleUseNode` que modela a ocorrência de um identificador único, previamente declarado; e o `UseNodeList` que modela uma lista de ocorrências que compõe uma estrutura sintática única, por exemplo, um nome de classe qualificado com o nome do pacote.

A interface `Scope` define métodos comuns a dois tipos de escopo: a classe `ScopeImpl` que modela escopos nos quais a ordem de ocorrência das declarações não importa, por exemplo, um corpo de classe; e a classe `BlockScopeImpl` que modela escopos nos quais a ordem das ocorrências das declarações deve ser considerada, por exemplo, em um corpo de método ou em um bloco.

O processo de criação do modelo pode ser dividido em três fases: pré-processamento, *parsing* e ligação.

5.1. Fase de pré-processamento

Nesta fase, todos tipos declarados no sistema são coletados. Primeiramente, todas subpastas do sistema são percorridas e para cada arquivo em cada pasta é criada uma referência-p. Depois, consultas SCQL extraem as declarações de tipo desejadas, as quais são indexadas pelo seu nome completamente qualificado. O algoritmo de pré-processamento é mostrado abaixo.

```
preProcessSystem(File d) {
  File[] subPack = Selecione todos subpacotes em d
  para cada package em subPack
     preProcessSystem(package);
  File javaFiles = Selecione todos arquivos Java em d
  para cada jFile em javaFiles
    Reference r = createReference(jFile,"#compilation_unit");
    Reference classSet[] =
        "VIEW TREE r AS TABLE(#class_declaration NOTCOMPOSING #class_declaration)";
    para cada reference em classSet
      Acrescente reference no índice de declaração de classes
    Reference interfaceSet[] =
        "VIEW TREE r AS TABLE(#interface_declaration NOTCOMPOSING #interface_declaration)";
    para cada reference em interfaceSet
      Acrescente reference no índice de declaração de interfaces
}
```

5.2. Fase de *parsing*

Nesta fase, todos os nós de declaração, escopo e uso são criados. O processo de *parsing* é aplicado em todas referências-p criadas na fase anterior. O processo é similar a um *parser* recursivo descendente, usando-se um método para cada não terminal relevante.

Cada método decompõe uma referência-p de entrada em seus componentes e verifica se nós de escopo, declaração ou uso podem ser criados. Nós de escopo são criados para cada item do sistema de arquivos e para estruturas sintáticas, como, *compilation_unit*, *type_declaration*, *method_declaration*, *block*, *local_variable_declaration*. Nós de escopo são contêineres para nós de declaração, de uso e de escopo aninhado. Nós de declaração são criados quando são analisadas estruturas como, *class_declaration*, *interface_declaration*, *method_declaration*, *field_declaration*. Por exemplo, no método que analisa uma *class_declaration*, um objeto `SingleDeclNode` é criado e adicionado no nó de escopo previamente criado a partir do respectivo arquivo. Nós de uso são criados quando analisa-se estruturas sintáticas que contém identificadores.

Consultas SCQL foram usadas para extrair informação da maior parte das estruturas sintáticas que não tinham nenhuma regra recursiva ou tinham regras recursivas simples. Por exemplo, considere analisar uma estrutura sintática *name* cuja regra está definida abaixo.

name	*::=*	*identifier*
	|	*name qualified_name*
qualifier_name	*::=*	*identifier*

```
name(Reference r) {
    Reference[] ref = "VIEW TREE r AS TABLE(#identifier i)";
    para cada referência em ref
       crie um nó de uso para esta referência
       coloque este nó no escopo atual
}
```

Uma maneira alternativa de se fazer o mesmo, usando templates, é mostrada abaixo.

```
template #name BaseName #{
    #identifier i
}#
template #name RecursiveName #{
    #name n #qualified_name qn
}#
template #qualified_name QualifiedName #{
    #identifier i
}#

name(Reference r) {
    BaseName bn = new BaseName();
    RecursiveName rn = new RecursiveName();
    if(bn.match(r))
       crie um nó de uso para a referência bn.i
       coloque este nó no escopo atual
    else if(rn.match(r))
       name(rn.n)
       qualifiedName(rn.qn)
}
qualifiedName(Reference r) {
    QualifiedName qn = new QualifiedName();
    if(qn.match(r))
       crie um nó de uso para a referência qn.i
       coloque este nó no escopo atual
}
```

Parece estar claro que a solução SCQL é muito mais simples e limpa. Contudo, SCQL pode não manipular algumas estruturas recursivas. Por exemplo, considere a regra *conditional_and_expression* da gramática de Java. Neste caso, a solução deve se basear em *templates*.

conditional_and_expression	*::=*	*inclusive_or_expression*
	|	*conditional_and_expression* `''&&''` *inclusive_or_expression*

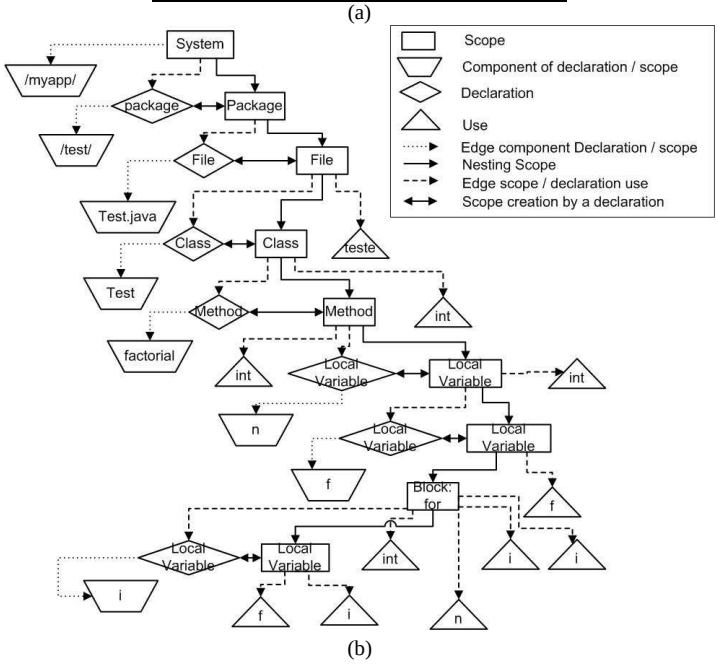
```
template #conditional_and_expression
         BaseConditionalAndExpression #{
    #inclusive_or_expression ioe
}#
template #conditional_expression
         RecursiveConditionalAndExpression #{
    #conditional_and_expression cae "&&"
    #inclusive_or_expression ioe
}#

conditionalAndExpression(Reference r) {
   BaseConditionalAndExpression bcae = new BaseConditionalAndExpression();
   RecursiveConditionalAndExpression rcae = new RecursiveConditionalAndExpression();
   if (bcae.match(r))
       inclusiveOrExpression(bcae.ioe);
   else if(rcae.match(r))
       conditionalAndExpression(rce.cae);
       inclusiveOrExpression(bcae.ioe);
}
```

Esta solução é similar a um *parser* recursivo descendente. Contudo, deve ser apontado que os *templates* são apenas uma ferramenta para criar este tipo de analisadores, mas também, podem ser aplicados com outras estratégias em geradores de código e transformadores de programas.

A Figura 2 apresenta o modelo resultante (b) para um programa Java simples (a).

```
package test;
public class Test {
  public int factorial(int n) {
    int f = 1;
    for(int i = 1;i <= n;i++)
       f *=  i;
    return f;
  }
}
```

(a)



(b)

Figura 2: Resultado da fase *parsing*

5.3. Fase de ligação

No início desta fase, todos nós do modelo já foram criados. Nesta fase, os nós de declaração serão ligados aos nós de uso e vice-versa. Esta fase é implementada com três *visitors*[Gamma et al., 1994]: `ExtendsLinkVisitor`, `DeclUseLinkerVisitor` e `QualifiedNameVisitor`, os quais não usam nem MetaJ nem SCQL, mas somente o modelo produzido até então. Isto mostra a utilidade de se ter facilidades de meta-programação embutidas em uma linguagem orientada a objetos. A Figura 3 mostra o modelo resultante para o programa apresentado na Figura 2.

O primeiro *visitor* é especializado em visitar cláusulas *extends* e *implements* para ligar os identificadores de classe às suas respectivas declarações. O seu algoritmo é mostrado abaixo.

```
ExtendsLinkerVisitor()
  para cada escopo de pacote na raiz do sistema
    para cada declaração de classe no escopo corrente
      se classe corrente tem cláusula extends
        // Pesquisa O(1) no índice gerado na fase de pré-processamento
        Node n = Obtenha a declaração da classe
        se(! n.isNull())
          ligue o nó de uso à sua declaração
      se classe corrente tem cláusula implements
```

```
Node use[] = Obtenha todas interfaces usadas na cláusula
Node decl[] = Obtenha as declarações das interfaces da cláusula
para cada nó em use
  ligue nó à respectiva decl
```

O segundo *visitor* liga declarações e seus usos que não são usos qualificados, por exemplo, uso de variáveis locais e acessos de variáveis de instâncias e chamada de métodos não qualificados. O modelo corrente é percorrido, em pré-ordem, dirigido pelos nós de escopo, e quando se encontra um nó não ligado e não criado a partir de um nome qualificado, é verificado se a respectiva declaração está no escopo corrente. Se está a ligação é feita, senão cria-se outro iterador e pesquisa-se no escopo acima na árvore em direção à raiz até se encontrar a declaração correspondente. O nó raiz de escopo contém todas as declarações pública do sistema, o que garante que todos nós serão ligados.

O terceiro *visitor* liga nós de uso correspondendo a nomes qualificados ocorrendo em em estruturas como, declarações de importação, chamadas de métodos e acessos a campos. Estas estruturas são representadas no modelo como nós `UseNodeList`. A respectiva declaração do elemento $i + 1$ da lista está no escopo criado pela respectiva declaração do elemento i da lista. O algoritmo é mostrado abaixo.

```
QualifiedNameVisitor()
  para cada escopo no sistema raiz
    se (uso corrente é qualificado)
      Node decl = null;
      para cado uso na lista de usos corrente
        se (decl == null)
           decl = Pesquise a declaração do primeiro uso da lista
        senão
           decl = Pesquisa a declaração do uso corrente no escopo aberto por decl
        ligue o uso corrente com decl
```

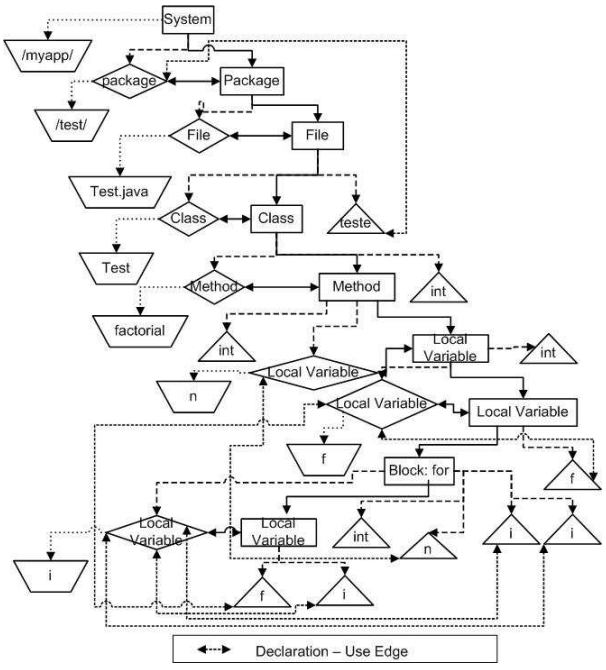


Figura 3: Resultado da Fase de Ligação

6. Estudo de caso: refatorações

Nesta seção, as ferramentas MetaJ e SCQL são comparadas entre si. A metodologia usada foi implementar duas refatorações, *Rename Local Variable (RLV)* e *Self Encapsulate*

Field (SEF), ambas com MetaJ e com SCQL. As refatorações foram aplicadas a três programas artificiais. Foram feitas 3 comparações: qualitativa, sobre o estilo de especificação oferecido pelas abordagens; quantitativa em relações às métricas de projeto dos respectivos refactorings; quantitativa em relação ao desempenho da aplicação dos refactoring a programas com determinadas características.

6.1. Análise qualitativa

O seguinte método foi extraído da implementação da refatoração RLV escrita com MetaJ. O fragmento é chamado quando existe colisão de nome entre o novo nome da variável local com o nome de uma variável de instância.

```
void addThisToFieldAccess(String var, Reference ctx) {
   Iterator it = ctx.getIterator();
   while(it.hasNextIn()){
      it.nextIn();
      Reference r = it.get();
      if(r.isTypeOf("java.#postfix_expression") && r.toString().startsWith(varName))
         r.set("this." + r.toString());
   }
}
```

O mesmo método escrito com SCQL é mostrado abaixo.

```
String findFieldAcc =
 "VIEW TREE context AS TABLE (#postfix_expression pe
 FILTERED BY pe.toString().startsWith(outer.varName))";
void addThisToFieldAccess(String var, Reference ctx) {
   QueryFactory qf = SCQL.createQueryFactory("java");
   qf.add("varName", var); qf.add("context", ctx);
   ResultSet rs = qf.createQuery(findFieldAcc).getResultSet();
   while(rs.next()){
      Reference r = rs.getReference("pe");
      r.set ("this." + r.toString());
   }
}
```

A abordagem MetaJ define um caminhamento no código que filtra uma estrutura sintática desejada e aplica uma ação desejada sobre ela.

A abordagem SCQL define declarativamente todas as estruturas sintáticas desejadas. A ação é depois aplicadas sobre as estruturas selecionadas.

Quando esta situação se escala para especificações maiores, a abordagem SCQL mostra separadamente as estruturas selecionadas e as ações executadas. O código SCQL se apresenta mais modularizado e legível, além de permitir a reutilização de consultas para seleções específicas. Os loops MetaJ são mais difíceis de serem reutilizados pois as ações estão embutidas no caminhamento.

6.2. Análise de métricas de projeto

As métricas de projeto coletadas sobre o código das refatorações, apresentadas na Figura 4, são: o número de linhas de código, o número de classes, o número de declarações de métodos e o número de declarações de campos.

O objetivo principal desta análise é verificar o tamanho das implementações. A diferença é mais notada na implementação da refatoração SEF, onde a abordagem SCQL têm a metade do tamanho. Cada *template* usado em MetaJ foi contado como uma classe. Cada meta-variável foi contada como uma variável de instância, com respectivos métodos get/set. Porém, o número de linhas que foi considerado para o *template* não foi igual ao da classe gerada, mas sim do *template* propriamente dito. A idéia é medir o esforço do programador. O uso intenso de *templates* para a refatoração SEF explica porque o número de campos para a refatoração SEF escrita com MetaJ é muito maior que com SCQL.

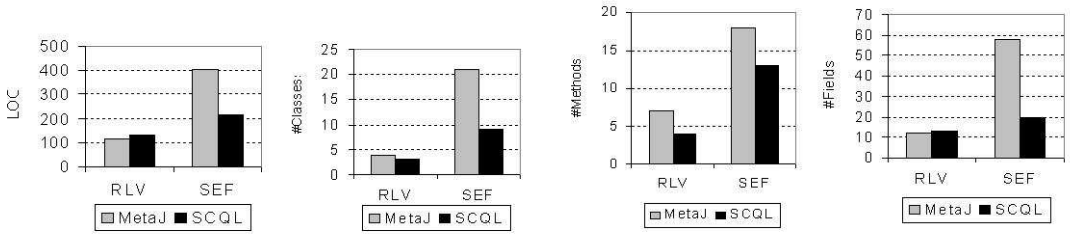


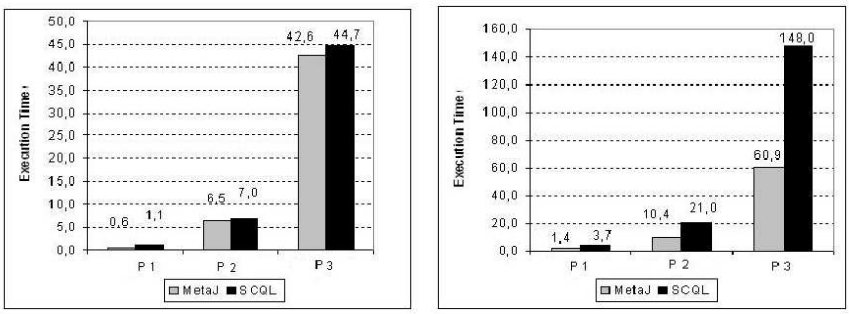
Figura 4: Métricas para os códigos das refatorações

6.3. Análise de desempenho

A comparação de desempenho das refatorações foi feita sobre três programas artificiais. A tabela seguinte mostra as principais características dos programas. LCM é o número de linhas de código por método; #C, #M, #F são os números de classes, métodos e campos, respectivamente; AAT é o número total de acessos e atribuições seja a variáveis locais ou variáveis de instância, em cada método; AAIV é o número de acessos e atribuições à variável de instância que será encapsulada pela refatoração SEF, em cada método; e AALV é o número de acessos e atribuições à variável local que será renomeada pela refatoração RLV, em cada método. Todos os métodos de uma classe são idênticos, de tal forma que a refatoração SEF afetará todos os métodos. Os 60 métodos nos programas *P1* e *P2* são idênticos. *P2* e *P3* têm a mesma estrutura, mas os métodos de *P3* são maiores.

	LOC	#C	#M	#F	LCM	AAT	AAIV	AALV
P1	383	1	10	10	36	55/24	2 / 1	4/2
P2	1888	1	50	10	36	55/24	2 / 1	4/2
P3	3715	1	50	10	71	110/42	5 / 2	7/3

Os parâmetros de entrada para a execução das refatorações foram tais que a refatoração RLV renomeia uma variável local declarada no meio de um método. O nome da variável local foi escolhido de tal forma que ele colida com o nome de uma variável de instância. O método escolhido para a refatoração RLV está localizado no meio da classe. Além disso, a refatoração SEF adiciona métodos get/set para uma variável de instância, de forma que exista uma variável local de mesmo nome no meio de todos os métodos. O objetivo destas escolhas foi simular uma média de esforço das aplicações das refatorações. Cada refatoração foi executadas cinco vezes sobre cada programa. Na Figura 5 os tempos médios de execução são mostrados. O desvio-padrão foi desprezível.



(a) (b)

Figura 5: Tempos de execução médios (a) Refatoração RLV (b) Refatoração SEF

Consultas e atualizações implementadas com SCQL, no caso da refatoração SEF, executaram dentro de um escopo específico a um método. Em uma implementação ingênua anterior, as consultas que calculavam produtos cartesianos em cinco conjuntos

eram feitas no escopo da classe inteira a cada vez, e o resultado foi um tempo de execução de cerca de dezesseis horas da refatoração SEF sobre Program4.

Pode-se perceber que o tempo de execução das refatorações com SCQL é razoavelmente aceitável em todos os casos da refatoração RLV. Porém, é necessária uma atenção especial à refatoração SEF que chega a ser duas vezes mais lenta.

7. Discussão

A abordagem apresentada para análise e manipulação de programas é baseada na definição de vários mecanismos para extrair informação sintática do código-fonte, os quais foram definidos considerando somente a sintaxe da linguagem fonte. Estes mecanismos podem ser utilizados para produzir modelos (orientados a objetos) mais expressivos e adequados a análises mais sofisticadas. Como mostrado na Seção 4, várias outras abordagens, tais como Genoa e SCA, incluiram mecanismos ad-hoc para construir uma base de conhecimento sobre o software sendo analisado, tornando fácil consultas sobre informações semânticas. Contudo, nossa abordagem é mais estratificada e reusável, se for considerada a construção de analisadores de programas para diferentes linguagens.

A abordagem apresentada mostrou que diferentes mecanismos, variando de algoritmos procedimentais iterativos até consultas e *templates* declarativos podem ser usados juntos em um mesmo meta-programa orientado a objetos, se beneficiando das práticas de projeto orientado a objetos. Os elementos imperativos da abordagem provaram ser mais eficientes, porém com restrições sobre a facilidade de uso e reuso. Os elementos declarativos são mais legíveis e fáceis de usar, mas tendem a ser computacionalmente mais caros. O uso de *templates* e consultas parece ser mais intuitivo que o uso de *visitors*, como por exemplo, em JJTraveler. A fase de *parsing* apresentada na Seção 5.2, se escrita com JJTraveler, requisitaria um *visitor* para percorrer a árvore sintática e a implementação de operações para todos os nós da árvores. Mesmo que o nó visitado não tivesse nenhuma informação relevante, deveria ser implementada uma operação nula. Se o nó visitado representar uma declaração seria necessário capturar informação adicional, tais como, identificadores, tipos, modificadores que requisitariam *visitors* específicos. Em MetaJ, isto é implementado diretamente com *templates*.

Contudo, deve ser notado que quando iteradores e *visitors* são necessários com nossa abordagem, MetaJ não oferece um mecanismo elegante para combinar *visitors* como em JJTraveler. Uma implementação da fase de *parsing* com Yacc teria as mesmas desvantagens de JJTraveler, sem contar a dificuldade para combinar *visitors* e a necessidade adicional de programar a construção da árvore sintática.

8. Considerações Finais

Este artigo apresentou o uso integrado de iteradores, *templates* e consultas para análise e manipulação de código-fonte. Estes mecanismos estão mais próximos de programadores convencionais que outras ferramentas baseadas em sistemas de reescrita. Equipes de desenvolvimento podem efetivamente considerar escrever meta-programas que ajudam analisar e manipular código-fonte durante o processo de desenvolvimento, porque as ferramentas apresentadas, MetaJ e SCQL, provaram ser viáveis e mais intuitivas.

Sobre a expressividade das ferramentas, o modelo de programas para MetaJ e SCQL é baseado somente na sintaxe da linguagem fonte. Para especificar consultas mais poderosas seria necessário incrementar o modelo com informação semântica. Contudo, nossa decisão de projeto simplifica a construção de plug-ins para outras linguagens, e o uso de SCQL mostrou ser útil e simples de usar em muitas situações.

A comparação de MetaJ e SCQL mostrou que SCQL é útil para produzir especificações mais compactas, modulares e reusáveis. Contudo, a implementação de SCQL ainda merece mais atenção em relação à otimização da execução da consulta, além de existirem situações onde o casamento de padrões e iteradores são necessários, indicando a necessidade de incluir mecanismos de composição de iteradores em MetaJ.

Referências

Consens, M., Mendelzon, A., and Ryman, A. (1992). Visualizing and querying software structures. In *Proc. of the Intl. Conf. on Software Engineering*, pages 138–156. ACM.

Cordy, J. R., Dean, T. R., Malton, A. J., and Schneider, K. A. (2002). Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837.

Crew, R. F. (1997). ASTLOG: A language for examining abstract syntax trees. In *Proc. of the USENIX Conference on Domain-Specific Languages*, pages 229–242.

Devanbu, P. (1999). GENOA - a customizable, front-end-retargetable source code analysis framework. *ACM TOSEM*, 8(2):177–212.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

Horwitz, S. (1990). Adding relational query facilities to software development environments. *Theoretical Computer Science*, 73:213–230.

Janzen, D. and de Volder, K. (2003). Navigating and querying code without getting lost. In *Proc. 2nd Intl. Conf. on Aspect-oriented Software Development*, pages 178–187.

Jarzabek, S. (1998). Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215.

Johnson, S. (1975). Yacc: Yet another compiler compiler. Technical report, Bell Telephone Laboratories.

Linton, M. (1984). Implementing relational view of programs. In *Proc. of ACM Software Engineering Symp. - Practical Software Development Environment*, pages 132–140.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Oliveira, A. (2004). MetaJ - An environment for metaprogramming in Java. *in portuguese*. Master's thesis, Federal University of Minas Gerais, Brazil.

Oliveira, A., Braga, T., Maia, M., and Bigonha, R. (2004). MetaJ: An Extensible Environment for Metaprogramming in Java. *Journal of Universal Computer Science*, 10(7):872–891.

Paul, S. and Prakash, A. (1996). A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3).

Sellink, M. P. A. and Verhoef, C. (1998). Native patterns. In *Proc. 5th Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society Press.

v. d. Brand, M. and et.al. (2001). The ASF+SDF meta-environment: A component-based language development environment. In *Computational Complexity*, pages 365–370.

van Deursen, A. and Visser, J. (2004). Source model analysis using the jjtraveler visitor combinator framework. *Software - Practice and Experience*, 34(14):1345–1379.